Planting Bugs: A System for Testing Students' Unit Tests

Samuel A. Brian Richard N. Thomas James M. Hogan Colin Fidge
School of Electrical Engineering and Computer Science
Queensland University of Technology
Brisbane, Qld. Australia
+61 7 3138 2736 +61 7 3138 9328 +61 7 3138 2870
s.brian@qut.edu.au r.thomas@qut.edu.au j.hogan@qut.edu.au c.fidge@qut.edu.au

ABSTRACT
Automated marking of student programming assignments has long been a goal of IT educators. Much of this work has focused on the correctness of small student programs, and only limited attention has been given to systematic assessment of the effectiveness of student testing. In this work, we introduce SAM (the *Seeded Auto Marker*), a system for automated assessment of student submissions which assesses both program code and unit tests supplied by the students. Our central contribution is the use of programs seeded with specific bugs to analyse the effectiveness of the students' unit tests. Beginning with our intended solution program, and guided by our own set of unit tests, we create a suite of minor variations to the solution, each seeded with a single error. Ideally, a student's unit tests should not only identify the presence of the bug, but should do so via the failure of as small a number of tests as possible, indicating focused test cases with minimal redundancy. We describe our system, the creation of seeded test programs and report our experiences in using the approach in practice. In particular, we find that students often fail to provide appropriate coverage, and that their tests frequently suffer from their poor understanding of the limitations imposed by the abstraction.

Categories and Subject Descriptors
J.1 [**Computer Applications**]: Administrative Data Processing – *Education.* K.3.1 [**Computing Milieux**]: Computer Uses in Education – *Computer-Assisted Instruction (CAI).* K.3.2 [**Computing Milieux**]: Computer and Information Science Education – *Computer Science Education.*

General Terms
Measurement, Languages, Verification.

Keywords
Automated Assessment; Technology in Education; Unit Testing.

1. INTRODUCTION
IT educators have long attempted to automate grading of student programming assignments, with tools emerging as early as the late 1960s [1]. Most of these tools, including many in current use, focus on the assessment of (often small scale) student programs, particularly those which might appear in a CS1 or CS2 course. As industry has demanded greater expertise in software engineering from IT graduates, so educators have given greater attention to the important skill of unit testing – especially automated testing in the context of agile development. Yet while canonical unit tests are often employed as a means to ensure the correctness of student program code, far less attention has been given to assessing the quality of the tests produced by the students themselves. At the Queensland University of Technology (QUT) we have developed SAM (the *Seeded Auto Marker*), a tool that allows us to test both student programs *and* their associated tests, scoring the programs with respect to the bugs identified by our own test cases, and scoring the student test cases with respect to their success in detecting errors intentionally introduced into an otherwise correct model solution. In this paper, we describe our approach, the tool itself and how it is used in practice to assess programs and tests.

This general approach has been used at QUT since 2009 in the third software development unit students encounter in the bachelor degree, with SAM, the present tool, introduced in 2014. The unit, entitled *Software Development*, covers modern programming practices, following the introduction of CS1/CS2 level material in the pre-requisite units. Java is the programming language used, complementing earlier studies of Python and C#. Software Development is intended to produce skilled developers capable of contributing to a software engineering team, as required in a subsequent unit that covers agile software development practices. In 2014 over 250 students completed the unit.

The topics covered in Software Development include the core Java language, application programming interfaces (APIs) and the Java library, unit testing (JUnit) and test-driven development (TDD), source control systems, design patterns and refactoring, simple graphical user interface programming and event handling, database connectivity, and simple concurrency. This wide range of topics is taught from the unifying perspective of programming in the large as a software development professional, and distinguished from earlier CS1/CS2 material.

Students complete two major programming assessment tasks in this unit and both are assessed using the testing tool. The first assignment is completed individually and involves writing a small object-oriented program to solve a specified problem. In 2014 this was a simple simulation of a water release system for a dam. Students were provided with a framework for the assignment and were required to implement two classes that conformed to the interface specifications provided. This approach helps to reinforce the concept of abstraction [2] and simplified the design of the testing tool [3]. Students were also required to develop a suite of JUnit tests for their code. Student submissions include their code for the required classes and the JUnit tests for those classes.

The second, larger, assignment is completed in pairs and involves some GUI and database programming. Students are again provided with an initial framework for the assignment and with

ITICSE'15, July 6–8, 2015, Vilnius, Lithuania.

ACM 978-1-4503-3440-2/15/07…$15.00.
DOI: http://dx.doi.org/10.1145/2729094.2742631

interfaces to which their classes must conform. In the second assignment, students are meant to follow a test-driven development approach to writing their software.

In 2014 the second assignment was to write a simulation of a car parking lot, monitoring the flow of cars into and out of the lot and the time they spent parked. The simulator had to cater for issues such as different sized parking spaces (e.g., car and motorcycle); only allowing cars to enter the car park if a parking space was available; allowing vehicles to queue and wait for spaces but with a finite length of queue and finite patience that drivers were willing to wait. For more detail about how these assignments are managed see our previous publication, *What vs. How: Comparing Students' Testing and Coding Skills* [4].

In both assignments, the effectiveness of unit tests carried a significant weight of the overall marks. For the first assignment, effectiveness of unit tests was worth 40%, the implementation (passing instructor-provided unit tests) was 35%, and code and documentation quality was 25%. In the second assignment the model implementation was worth two-thirds and the GUI implementation was worth one-third of the total mark. For the model implementation, effectiveness of unit tests was worth 25% of the mark, implementation 30%, code quality 15%, and pair process 30%. This means that the ability to effectively implement unit tests contributed to 13% of the students' final grade in the unit, which mapped well to the intended learning outcomes.

This paper is organised as follows. The extensive history of automated marking systems for programming assignments is reviewed briefly in Section 2. In Section 3, we describe our system and the marking process in detail, leading into Section 4, in which we consider the crucial question of buggy program code as a test case for unit tests. This section includes a number of examples of the choice of defect, and the results obtained. We conclude in Section 5 with further discussion of the approach and consideration of future work.

2. PREVIOUS WORK

Automated program grading tools fall into two main categories: tools that attempt to evaluate the correctness of submitted programs; and tools that attempt to evaluate the quality of submitted programs [5]. In a number of cases tools span both categories. The focus of our work is on the first category, evaluating the correctness of programs, though our work differs markedly from its predecessors in its approach to assessing unit tests.

Many of these tools, for example BOSS2 [6], CourseMarker [7] and Oto [5] focus just on the submitted program, testing its correctness using functional and/or unit testing. This may be appropriate for introductory subjects that wish to focus on programming and problem solving, but it is limiting for subjects that require students to implement their own test suites.

Other tools, such as ASSYST [8] focus on functional testing, determining if the submitted test data provides adequate test coverage of the submitted program. This is suitable for small programs but for larger programs it is useful to be able to identify the particular methods that are causing errors rather than just identify an incorrect result.

Tools such as AutoGrader [3], Marmoset [9] and Web-CAT [10] assess unit tests submitted with a program but limit their testing to comparing the results of student submitted unit tests against those of the instructor-supplied tests when run against the submitted program. This is useful in finding errors in the student program that are not identified by the student's own tests but does not evaluate the overall effectiveness of student-supplied tests in identifying other defects.

Like ProgTest [11], our SAM tool not only compares student and instructor tests when run against the student's program, but also runs the student tests against a sample solution. This allows the students' tests to be assessed more thoroughly, particularly in cases where students submit unit tests for methods not fully implemented in their programs. This corresponds to the expectation of TDD that tests will be written first. While the majority of students are expected to complete all aspects of assignments, there will inevitably be some students whose submissions are incomplete. Valid tests submitted by the student may fail on their own program code, but pass when run against the sample solution.

A key feature of our tool is that it uses a suite of erroneous programs to more fully assess the student's unit tests. Each erroneous program contains a *single error* that should be caught by at least one of the unit tests. (The erroneous programs are generated by the instructor guided by our own unit tests, which are produced by following TDD practices while implementing the sample solution.) The student's unit tests are then executed against each of the erroneous programs, and the tool determines the number of errors found by their test suite. This provides a rigorous evaluation of the student's unit tests as the tool can determine how many known errors were identified.

Several studies, as reported by Buffardi and Edwards [12], have identified that students are reluctant to adopt TDD. By placing a strong emphasis on TDD in lectures and laboratory sessions, by allocating a significant percentage of marks to writing unit tests, and by rigorously evaluating the effectiveness of unit tests, we are trying to encourage students to adopt TDD and its practices. We have found that the unit tests submitted by students in the second assignment are markedly improved compared to those submitted in the first assignment [4]. This does not guarantee that students are following TDD, but does at least demonstrate an improvement in their ability to effectively test their programs, which in itself is a useful outcome for a unit intended to produce skilled software developers. SAM provides a mechanism to allow unit tests to be rigorously evaluated without placing too large a burden on the academic staff marking the assignments.

3. TESTING TOOL

Our approach was to automate as much of the assignment marking process as was feasible, leaving only assessment of code quality to manual intervention—though even here the process was supported. The four automated steps are extraction, compilation, execution and results processing.

The first step is to extract the students' source code files from the submitted zip archives. A basic check of the presence of the required files is then performed. Submissions that failed this check often contained misnamed files or an unexpected file hierarchy, and were manually reorganised.

The compilation of each student's source code can be performed in two ways. One method is to compile all the student's source code together, just as the student would have on their own computers, which is required if the assignment specification is flexible in how it allows the students to develop their solutions. This method, however, does not detect violations of the assignment's API specifications—potentially causing run-time errors later where the problem is harder to find.

An alternative method is to compile the source files individually against the pre-compiled solution, detecting API problems at compile time. A common violation of the API specification is the students' introduction of extra public variables or methods, which is only a minor issue if the public fields are used by classes that are under the students' control only, but is a major problem when student unit tests that expect these public fields are run against the marker's solution which does not have them [4]. In these cases, the failed compilation logs allow the offending unit tests to be identified and removed from students' code. Also, a stubborn marker can look at the student's code to see if it is possible to modify the unit test to use public methods or fields from the published interface. This allows part marks to be given for identifying valid test cases, even if the unit test is implemented incorrectly. Deleting the offending unit tests or modifying them requires manual editing of the student's submission, but deleting offending tests takes much less time than attempting to modify tests. These problems arise frequently even though students are provided with a tool to check for compile-time errors and the structure of their code archives before submission.

The execution step involves running sets of unit tests and piping the output to files. Figure 1 shows how a student's tests and implementation are separated and used in three different kinds of tests: (i) the student's implementation against the solution unit tests assessing the correctness of the student's program code for the implementation mark; (ii) the solution implementation against the student's unit tests—ensuring basic correctness of the student's unit tests; and (iii) the broken implementations against the student's unit tests—assessing the effectiveness of student-supplied tests in finding defects. Both steps (ii) and (iii) are used to calculate the unit test mark.

Some issues may inadvertently emerge in the testing of broken implementations, with student tests relying on a correct implementation for a loop condition disrupted by the planted bug. Timeout conditions are thus an essential part of the process.

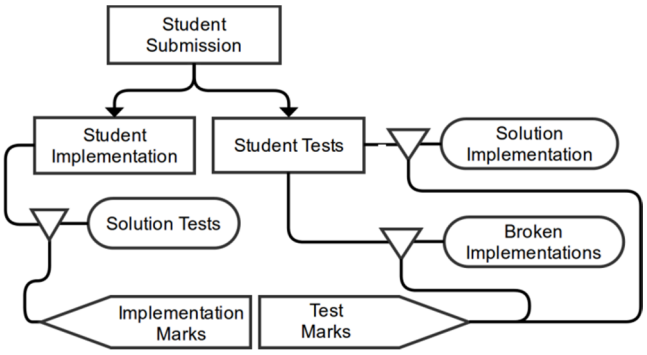

Figure 1. Student Submission Execution Process

The final automated step is the processing of the test results. The logs of unit test outputs are scraped for the raw results: the number of tests passed out of the tests performed. The mark given for students' implementation code is simply proportional to the fraction of our solution tests passed.

Calculating the mark for the students' unit testing code is more involved as a successful test suite will have at least one failed unit test for each broken implementation (true positives), but the students' tests that incorrectly fail the solution implementation (false negatives) must also be considered. A 'broken' implementation is considered to have been detected if a student's test suite shows more failed tests than when the student's tests are run against the solution implementation, and a mark can then be awarded that is proportional to the fraction of broken implementations detected.

4. EXAMPLE SCENARIO

In the 2014 car park assignment, students had to implement four classes to complete the model: a `CarPark` class, a `Vehicle` abstract class, and `Car` and `MotorCycle` classes that extend `Vehicle`. The required test suite consisted of `CarParkTests`, `CarTests`, and `MotorCycleTests` classes, where the `Vehicle` class is tested indirectly through the subclasses.

For instance, one student's submitted implementation of the model passed 21 of 56 unit tests from the marker's solution `CarParkTests`, 7 of 9 from `CarTests`, and 48 of 53 from `MotorCycleTests`. By weighting these results to 7, 1, and 4 marks respectively, the total implementation mark is calculated as shown in Table 1 as 7.026/12.

Table 1. Solution Tests vs. Student Implementation

Test Class	Tests		Marks	
	Passed	**Out Of**	**Score**	**Out Of**
`CarParkTests`	21	56	2.625	7
`CarTests`	7	9	0.778	1
`MotorCycleTests`	48	53	3.623	4
		TOTAL:	7.026	12

With our solution implementation, the student submission's test suite passed 19 of 22 tests in `CarParkTests`, 0 of 33 in `CarTests`, and 30 of 30 in `MotorCycleTests`, as per Table 2.

Table 2. Student Tests vs. Solution Implementation

Test Class	Passed	Out Of
`CarParkTests`	19	22
`CarTests`	0	33
`MotorCycleTests`	30	30

The anomaly of every unit test in `CarTests` failing the solution implementation suggests some fundamental misunderstanding of the assignment specification. In this case, two `Car` objects were constructed before each `CarTests` unit test was run, the second with an invalid argument (the `-1` below):

```
@Before
public void setUp() throws Exception {
  carTest = new Car("C1", 1, true);
  carTest2 = new Car("C2", -1, false);
}
```

This invalid argument caused an exception to be thrown by our solution implementation, but not the student's own implementation which failed to check it (causing at least one of the failures against the solution `CarTests` where it passed 7 of 9 tests).

The tests that failed in the results described in Table 2 are false negatives, and were used as a reference to compare the results of the submission's tests run against our broken implementations. Two suites of broken implementations were constructed for marking: *brokenVehicles* and *buggyCarParks*. As the students were not instructed which test class (`CarTests` or `MotorCycleTests`) would be used to test broken implementations in *brokenVehicles*, both were evaluated. Three scenarios occurred when determining if a bug in a broken implementation had been detected by the students' tests.

In a 'broken' variation of the solution implementation titled *brokenVehicles_Fails_Vehicle_enterParkedState_NoThrow_IncorrectState*, one exception check was removed from the `enterParkedState` method. When run against the student's `MotorCycleTests`, it resulted in one more failure (29/30) than when the student's tests were run against our solution implementation (30/30), meaning that the student's unit test suite successfully detected the bug (as shown in Table 3).

Table 3. Test results that show more failures than reference (bug detected)

Test Class	Passed	Out Of	Reference	Detected
`CarTests`	0	33	0	*No*
`MotorCycleTests`	29	30	30	*Yes*

Another of our broken implementations is *brokenVehicles_Fails_CarConstructor_IncorrectCarSize* which fails to correctly set the "small car" Boolean flag which is passed as an argument to the `Car` constructor. An equal number of failures occurred with this broken implementation as with the reference results (shown in Table 4), which means that the student's tests failed to detect the bug.

Table 4. Test results that show failures that equal reference (bug not detected)

Test Class	Passed	Out Of	Reference	Detected
`CarTests`	0	33	0	*No*
`MotorCycleTests`	30	30	30	*No*

Lastly, our *brokenVehicles_Fails_Vehicle_Constructor_NoThrow* broken implementation resulted in *fewer* failures (Table 5) because the exception check that is absent is the one that caused all of the submission's `CarTests` to fail previously. The bug is considered undetected by the student when this scenario occurs.

Table 5. Test results that show fewer failures than reference (bug not detected)

Test Class	Passed	Out Of	Reference	Detected
`CarTests`	33	33	0	*No*
`MotorCycleTests`	30	30	30	*No*

By applying this process to all of the broken implementations, the student submission's tests were found to detect 14 of 27 bugs from *brokenVehicles*, and 4 of 17 bugs from *buggyCarparks* (including 1 broken `Car` subclass). These results are weighted to 2 and 3 marks respectively, resulting in the marks in Table 6.

Table 6. Student Tests vs Broken Implementations

Broken Implementation Suite	Broken Implementations		Marks	
	Detected	Out Of	Score	Out Of
brokenVehicles	14	27	1.037	2
buggyCarParks	4	17	0.706	3
		TOTAL:	1.743	5

When the automatic process is complete and a submission has been given a mark for its implementation and unit tests, the manual marking step must be performed. As part of SAM, manual marking is aided by providing a comment box and mark entry boxes for additional criteria such as code quality and GUI functionality, quality, and testing, from which the total mark can be calculated (see Figure 2). The tool allows a compressed archive to be created of the whole directory containing a student's compilation logs, raw test results, and a marking rubric pre-filled with the marks and comments, all of which may be returned to the student.

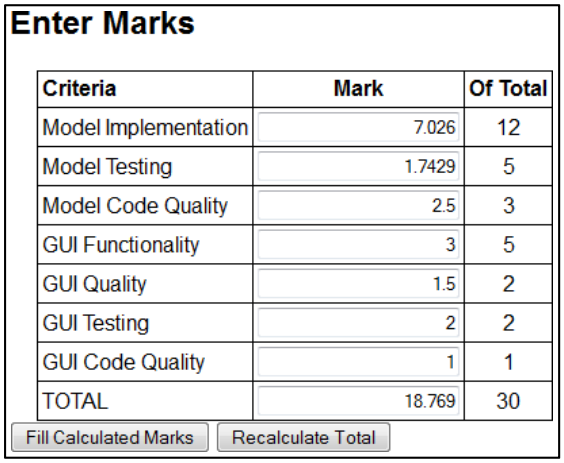

Figure 2. Final Mark Calculation Screen

5. BUGGY IMPLEMENTATIONS

This section provides two examples of deliberate errors used to assess the effectiveness of student-submitted unit test suites. The examples cover the simple case of a unit test that depends on just the method being tested and a more complex case where the unit test calls more than one method. Each example shows the "buggy" method, a unit test that detects the error, and a student test for the method that failed to detect the error.

In the simple case where a unit test only calls the method being tested, a failure is a clear indication that the bug has been identified. The constructor of the `Vehicle` class was required to throw an exception if the arrival time argument was zero or negative, meaning that the vehicle arrived *before* the first minute the car park was open. The commented-out condition below is the one we removed to 'break' the method.

```
public Vehicle(String vehID, int arrivalTime)
  throws VehicleException  {
//if (arrivalTime <= 0) {
// throw new VehicleException("Vehicle " +
//  vehID + ": arrival time must be strictly
//  positive");
//}
 this.vehID = vehID;
 this.arrivalTime = arrivalTime;
 // ... other initialisation
}
```

The following unit test will catch this error because the `Vehicle` constructor no longer throws the `VehicleException` that is expected by the test.

```
@Test (expected = VehicleException.class)
public void testCarConsExceptionZero()
  throws VehicleException {
 Car c = new Car("brokenZero", 0, true);
}
```

The following unit test is an example of a student test that did not catch the error. It is a simple error: the student did not anticipate the exception to be thrown.

```
@Test
public void testCarZeroIsSatisfied()
  throws VehicleException {
 Car c = new Car("XYZ012", 0, true);
 assertFalse(c.isSatisfied());
}
```

Here, our buggy code will detect the incorrect unit test as the test will *not* fail when the tool expects it to.

A more complex situation occurs when a unit test calls two or more methods. When evaluating the test it is difficult to determine which buggy method caused the test to fail. The `Vehicle` class has a method `exitParkedState`, which causes the vehicle to exit the car park after earlier parking successfully. One of our buggy versions of this method introduced an off-by-one error. The second condition, underlined below, should be < and not <=.

```
public void exitParkedState(int departureTime)
throws VehicleException {
  String msg = "";
  if (this.inQueue || !this.parked) {
    msg = " in queue or otherwise not parked";
    throw new VehicleException("Vehicle " +
                               this.vehID + msg);
  }
  if (departureTime <= this.parkingTime) {
    msg = ": departure time cannot be earlier than";
    msg += " parking time";
    throw new VehicleException("Vehicle " +
                               this.vehID + msg);
  }
  this.departureTime = departureTime;
  this.parked = false;
}
```

The following unit test will catch the error because the buggy code will throw an exception when `exitParkedState` is called, causing an error in the JUnit runner as the Exception is not expected,indicating to the tool that the bug was identified.

```
@Test
public void testExitParkedValidDepartOnParking()
throws VehicleException {
  this.testMC.enterParkedState(PARK_ON_ARRIVAL,
                               DURATION);
  this.testMC.exitParkedState(PARK_ON_ARRIVAL);
  assertTrue("Park=Departure",
             !this.testMC.isParked());
  assertEquals("Park=Departure", PARK_ON_ARRIVAL,
               testMC.getDepartureTime());
}
```

The following unit tests are examples of student tests that did not catch this error. In this case the student had thought to check for off-by-one errors, but was unable to translate this idea into a correctly structured test with the appropriate logic, with the result that their tests did not identify the real bug.

```
@Test (expected = VehicleException.class)
public void testExitParkedValidDepartOnParking()
throws VehicleException {
  this.testMC.enterParkedState(PARK_ON_ARRIVAL,
                               DURATION);
  this.testMC.exitParkedState(PARK_ON_ARRIVAL);
}

@Test
public void testExitParkedValidDepartOnParking()
throws VehicleException {
  this.testMC.enterParkedState(PARK_ON_ARRIVAL,
                               DURATION);
  this.testMC.exitParkedState(PARK_ON_ARRIVAL+1);
  assertTrue("Park=Departure",
             !this.testMC.isParked());
  assertEquals("Park=Depart", PARK_ON_ARRIVAL+1,
               testMC.getDepartureTime());
}
```

In this situation, the tool can determine that the student has not identified the incorrect logic of the broken program, and this is correctly reflected in the mark calculated by the tool. However, one challenge is that the student's test calls *two* methods – `enterParkedState` and `exitParkedState`. When using the instructor-supplied tests to assess the student's program, the tool cannot determine *which* of the methods caused the error, and so cannot provide detailed feedback beyond recording the fact that the test did not detect the error.

6. CONCLUSIONS

In this work, we have introduced a new tool, SAM, developed to assess automatically student programming assignments and their associated unit tests. In contrast to many previous approaches (see Section 2) our approach pays particular attention to the problem of unit testing, and in particular the problem of assessing the effectiveness of unit tests in detecting known defects. Following the strategy we introduced earlier [4], we assess unit tests against a suite of variations on the model solution, otherwise correct implementations seeded with a single known defect.

Use of the tool has provided a number of insights into the problems experienced by intermediate students transitioning from CS1 and CS2 sized exercises to those approximating contributions needed in a professional project. Among the more important of these themes are:

- Weak test coverage overall;
- Misconceptions about the role of unit testing and the primacy of the object abstraction; and
- Diffuse and redundant test cases.

As we discussed previously [4], however, the approach does lead to significant improvements in performance between the first and second assignments in the unit. The use of SAM has allowed far better feedback to be provided, and fewer manual interventions in the marking process. Improvements to the system for 2015 will include a more sophisticated submission checking system, placing the responsibility for structural correctness on the students and reducing the tedious manual fixes that have been required to date.

7. AVAILABILITY

The SAM program is available at https://bitbucket.org/samuelbr/automarker-inb370 as a Python script with a web interface. The tool can be configured for Java programming assignments that are structured as described in this paper.

8. ACKNOWLEDGMENTS

We wish to thank Dr. Andrew Craik, now at IBM Canada, for developing the original assignment marking scripts upon which SAM has been based. We would also like to thank Lawrence Buckingham (QUT) for his contributions to the testing and development of the original marking scripts and SAM.

9. REFERENCES

[1] Hext, J. B. and Winings, J. W. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12, 5 (May 1969), 272-275.

[2] Schmolitzky, A. "Objects first, interfaces next" or interfaces before inheritance. In *Proceedings of the Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (Vancouver, BC, Canada, 2004). ACM.

[3] Helmick, M. T. Interface-based programming assignments and automatic grading of java programs. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education* (Dundee, Scotland, 2007). ACM.

[4] Fidge, C., Hogan, J. and Lister, R. What vs. How: Comparing Students' Testing and Coding Skills. In *Proceedings of the 15th Australasian Computing Education Conference (ACE 2013)* (Adelaide, Australia, January, 2013). Australian Computer Society, Inc.

[5] Tremblay, G., Guerin, F., Pons, A. and Salah, A. Oto: A Generic and Extensible Tool for Marking Programming Assignments. *Software Pract. Exper.*, 38, 3 (March 2008), 307-333.

[6] Joy, M., Griffiths, N. and Boyatt, R. The boss online submission and assessment system. *J. Educ. Resour. Comput.*, 5, 3 (September 2005), 2.

[7] Higgins, C., Hegazy, T., Symeonidis, P. and Tsintsifas, A. The CourseMarker CBA System: Improvements over Ceilidh. *Education and Information Technologies*, 8, 3 (September 2003), 287-304.

[8] Jackson, D. and Usher, M. Grading student programs using ASSYST. In *Proceedings of the 28th SIGCSE technical symposium on Computer science education* (San Jose, California, USA, 1997). ACM.

[9] Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K. and Padua-Perez, N. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education* (Bologna, Italy, 2006). ACM.

[10] Edwards, S. H. Teaching software testing: automatic grading meets test-first coding. In *Proceedings of the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Anaheim, CA, USA, 2003). ACM.

[11] Souza, D. M. d., Maldonado, J. C. and Barbosa, E. F. ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In *Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training* (May 22-24, 2011). IEEE Computer Society.

[12] Buffardi, K. and Edwards, S. H. Exploring influences on student adherence to test-driven development. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (Haifa, Israel, July 3-5, 2012). ACM.